Models of Horn Formulas are Enumerable at Nearly Linear Delay

Johann Brault-Baron

Models of Horn Formulas are Enumerable at Nearly Linear Delay

<Johann.Brault-Baron@unicaen.fr>

Abstract

The Unique satisfiability problem of Horn formulas was proved to be quadratic by Minoux (1992) [3]. Using ideas based on those of [3], Berman, Franco, and Schilpf (1995) [1] proved it to be nearly linear. We simplify the presentation of their algorithm and adapt it slightly in order to perform enumeration of the solutions at a delay that *is* their unique solution decision time, i.e. $\mathcal{O}(\alpha(|F|)|F|)$ where $|F|$ is the formula size and α is the inverse Ackermann function, or (at choice) $\mathcal{O}(n \log n + |F|)$ where n is the number of variables of F.

Keywords: propositional Horn formula, enumeration algorithm, nearly linear time

Introduction

Counting the models of a propositional Horn formula is a #P-Complete [5] problem, therefore not tractable. Nevertheless, the unique satisfiability problem, which can be considered as a (very) limited way of counting, i.e. it answers one of $\{0, 1, 2 \text{ or more}\}$, was proved to be nearly linear by [1]. We can generalize this "bounded counting" to the problem of counting *up to* k, for any fixed k, that returns a value in the set $\{0, 1, \ldots, k-1, k \text{ or more}\}$.

This paper provides an algorithm that extends the unique solution decision algorithm of [1], and performs exhaustive model enumeration at nearly linear delay. A straightforward corollary of our result is that counting up to k can be done in time $\mathcal{O}(k\alpha(|F|)|F|)$, with production of the corresponding solutions in the same time.

Organisation of this Paper

We present the algorithm in three steps. First of all, we give an overview of the algorithm (algorithm 1), with the logical argument (called *Trick1*) proving its correctness, and explaining the main idea. Nevertheless, this algorithm, if implemented poorly, has a $\mathcal{O}(n|F|)$ delay, which is the complexity of the naive algorithm. In sections 2 and 3, we give a more explicit writing of the algorithm on its critical part, with more implementation details, that allow to reach the expected complexity. Each section has its main argument, denoted respectively by *Trick1*, *Trick2*, and *Trick3*, that are *arguments on logic* that are the key points leading to properties of correctness and/or complexity. Note

that algorithms 2 and 3 below only explicitly describe the critical part of the algorithm and are justified resp. by *Trick2* and *Trick3*.

Preliminaries

We assume the reader is familiar with propositional logic. In the whole document, F will be a Horn CNF formula on n variables. Since F is assumed in CNF, we will see it as a set of clauses, which are seen as sets of literals, we therefore will use set notations accordingly, e.g., $\emptyset \in F$ means "F contains an empty clause." We define $F(l) = \{C \in F \,|\, l \in C\}$ and $F[l] = \{C \setminus \{\bar{l}\} \,|\, C \in F \setminus F(l)\}$. $F[l]$ is the formula obtained by setting l to 1 (true) in the formula and simplifying it accordingly. We assume no formula contains a tautological clause.

We use the RAM model (see [2]). In this model, an input of size N is a sequence of N integers identified to their binary representation, each one in the interval $[0, N[$ and contained in N registers of size $\lfloor \log(N-1) \rfloor + 1$ (the size of the binary representation of $N-1$) or $\theta(\log N)$.

As suggested by [1], we can see the formula as a *bipartite graph* where one of the vertex sets is the set of literals, and the other is the set of clauses; in this graph there is an edge between a literal and a clause iff the literal belongs to the clause. With this model, as in [1], the size $N = |F|$ of a formula is the total number of variable occurrences.

1. A First Approach

1.1. Algorithm Overview

This section starts by stating the adopted search strategy, and shows how it leads to an algorithm, that is presented in algorithm 1. The main idea is the following: we want to perform a recursive search, but with no backtracking due to failure, i.e. the recursive enumeration procedure should always produce at least one solution before backtracking.

That is why, as a precomputation step, we perform *positive* unit propagation. This way, if the formula is satisfiable, then it is zero-valid (i.e. $(0, \ldots, 0)$ is a model); the algorithm terminates at this point otherwise. In algorithm 1, this is done in the procedure EnumHorn, lines 2–5. In the whole algorithm, models are considered as sets of literals, where every variable appears once. Lines 7-8 can therefore be read as "for every model of the zero-valid formula obtained after positive unit propagation, complete it with the positive literals that were propagated." Zero-validity is the key property that will be maintained as an invariant, and that guarantees satisfiability is maintained, which is *Trick1*.

We will perform enumeration of the models of this zero-valid formula with the recursive procedure EnH_0 in algorithm 1. We need to recursively get rid of variables *in a way that maintains zero-validity*. Since the considered Horn formula is guaranteed to be zero-valid, two different situations occur. In both situations, checking zero-validity is maintained for each recursive call is easy; correctness follows.

```
EnumHorn(F, 𝒱):
    S ← ∅
    while ∃x ∈ 𝒱 {x} ∈ F do
        F ← F[x]
        S ← S ⊎ {x}
    if ∅ ∉ F then
        for M ∈ EnH₀(F, 𝒱 ∖ S) do
            yield M ⊎ S
end
EnH₀(F, 𝒱):
    if 𝒱 = ∅ then
        yield ∅
    else if ∃x ∈ 𝒱 ∀y ∈ 𝒱 {¬x, y} ∉ F then
        for M ∈ EnH₀(F[¬x], 𝒱 ∖ {x}) do
            yield M ⊎ {¬x}
        if {¬x} ∉ F then
            for M ∈ EnH₀(F[x], 𝒱 ∖ {x}) do
                yield M ⊎ {x}
    else
        Let S = {x₁, ..., xₖ} a circuit of G(F)
        foreach xᵢ ∈ S do
            Replace xᵢ by x₁ in F
            Replace ¬xᵢ by ¬x₁ in F
        S ← S ∖ {x₁}
        for M ∈ EnH₀ (F, 𝒱 ∖ S) do
            if x₁ ∈ M then
                yield M ⊎ S
            else
                yield M ⊎ {¬x | x ∈ S}
        Undo the replacements in F
end
```

Algorithm 1: Basic algorithm.

The first situation is the case where affecting a given variable x to 1 does not make a positive unit clause appear, i.e. there is no y such that $\{\neg x, y\} \in F$. This case splits into two sub-cases. If $\{\neg x\} \in F$, then the case is easy to deal with. In the other case, x can be set indifferently to 0 or to 1 without affecting the zero-validity property of F, we can therefore make one recursive call on $F[\neg x]$ and one on $F[x]$.

The second situation is the case where, for every variable x, we can find y such that $\{\neg x, y\} \in F$. As a consequence, the *implication digraph* $G(F)$ of the formula F, essentially introduced by [3] and defined as $G(F) = \{x \to y \mid \{\neg x, y\} \in F\}$, has a circuit (of length at least two): we can therefore replace, in the formula, any variable appearing in the circuit, by a given variable of the circuit; in every model, they have the same value.

1.2. First Complexity Considerations

Consider algorithm 1. The precomputation is linear, we do not need to actually consider it. What is the cost of the instruction "Undo" on line 21? We can imagine that, every time a value is changed in memory, a backup save is made and pushed on some stack without affecting the complexity up to a constant factor; therefore undoing the changes costs at most as much as performing them, consequently we do not need to take it into account. This is the case of line 21, but not only: in particular, in the recursive calls of lines 5 and 8, $F[l]$ (where l is either $\neg x$ or x) is computed in time proportional to the "difference" between F and $F[l]$, i.e. the part of F that is removed, provided $F[l]$ is computed in-place. We therefore compute it in-place, but need to restore F after the recursive call. We can do it in the same time for the reason mentioned in previous paragraph. As a consequence, the first part (lines 2–9) is responsible for a linear delay $\mathcal{O}(|F|)$, i.e. the delay is linear if we exclude the time cost of everything else, that is to say the time needed for computing a circuit, and the variables replacements (lines 11-14).

```
1  EnH₀(F, 𝒱, visitList = ∅):
2  |  ...
10 |  else
11 |  |  if visitList = ∅ then
12 |  |  |  Take x ∈ 𝒱
13 |  |  else
14 |  |  |  x ← Peak(visitList)
15 |  |  |  Take x ∈ {y ∈ 𝒱 | {¬x, y} ∈ F}
16 |  |  while x ∉ visitList do
17 |  |  |  Push(visitList, x)
18 |  |  |  Take x ∈ {y ∈ 𝒱 | {¬x, y} ∈ F}
19 |  |  S ← ∅
20 |  |  while Peak(visitList) ≠ x do
21 |  |  |  y ← Pop(visitList)
22 |  |  |  S ← S ⊎ {y}
23 |  |  |  Replace y by x in F
24 |  |  for M ∈ EnH₀ (F, 𝒱 ∖ S, visitList) do  ...
29 |  |  Undo the changes of F
30 end
```

Algorithm 2: Incremental Circuit Building, where lines 11–15 of algorithm 1 have been substituted by lines 11–23, and lines 25–29 are just the same as the lines 17–21 in algorithm 1.

2. Incremental Circuit Building

We are now concerned with building a circuit of $G(F)$ (circuit, for short) *incrementally*, i.e. continuing each time what was begun before. In order to do so, we need to detail. To find a circuit, we only need to take any variable, and

then to follow a path until we find a variable that was already considered. Then this variable and all the variables that were considered after this first variable form a circuit. Building this circuit is done in algorithm 2, with the use of instructions on stacks, where $\emptyset$ denotes the empty stack:

Push(stk, val) adds a value on the top of the stack and returns the modified stack,

Pop(stk) removes the value on the top of the stack and returns this value, and

Peak(stk) returns the values on the top of the stack.

Trick2 is the following invariant of this algorithm: the set of visited variables and edges form a path of $G(F)$, maintained as a stack in the algorithm. This is the case since after a circuit contraction, the whole circuit becomes a single variable, which is on the top of the stack. It is not hard to see the first part of the algorithm (corresponding to lines 2–9 of algorithm 1) can only remove a variable that is on the top of the stack or unvisited, therefore maintaining the key property that the stack is a set of variables forming a path.

If we except the "Replace" instruction (line 23 of algorithm 2), this part is responsible for a linear delay. From now the goal is finding a way to avoid *actually* replacing the variables.

3. The Final Algorithm

3.1. Using an Union-Find Algorithm

We are concerned with having replacements made more easily. To do so, instead of replacing actually variables in the formula, we define equivalence classes, and search for equivalence class while considering a literal in a clause. This allows to make explicit use of the classical union-find algorithm.

Take a data structure representing a partition. The union-find procedures are the following operations:

Find(x) determines which set, given by its representative, x is in;

Union(x, y) merges the two sets represented respectively by x and y into a single set, returns the representative of this new set; and

MakeSet(x) makes a new set $\{x\}$, represented by x.

We are concerned by the time cost of n calls to Union and m calls to Find, where $m \geq n$. By a result of [4], this cost, depending on the union-find variant used, takes either $\mathcal{O}(n \log n + m)$ or $\mathcal{O}(\alpha(m).m)$ (see also [1] for details).

```
1  ...
16 while x ∉ visitList do
17     Push(visitList, x)
18     for C ∈ longCl(x) do
19         if firstVisit(C) ≠ ⊥ then
20             len(C) ← len(C) − 1
21             longCl(x) ← longCl(x) ∖ {C}
22             if len(C) = 1 then
23                 y ← Find(firstVisit(C))
24                 longCl(y) ← longCl(y) ∖ {C}
25                 shortCl(y) ← shortCl(y) ⊎ {C}
26         else firstVisit(C) ← x
27     Choose x ∈ {Find(y) ∈ V | {¬x, y} ∈ shortCl(x)}
28 last ← x; S ← ∅
29 while Peak(visitList) ≠ last do
30     y ← Pop(visitList)
31     S ← S ⊎ {y}
32     z ← Union(x, y)
33     longCl(z) ← longCl(x) ⊎ longCl(y)
34     shortCl(z) ← shortCl(x) ⊎ shortCl(y)
35     x ← z
36 ...
```

Algorithm 3: The Final Algorithm, where lines 16–23 of algorithm 2 have been substituted by lines 16–35.

3.2. The Final Algorithm

We present here the implementation details needed for getting the presented complexity; we have kept almost the same notations as those of [1]. The main point of this part is a logical argument (*Trick3*) of importance, based on Davis-Putnam resolution and on clause subsumption, that allows to guarantee two given sets of clauses are disjoint, which permits to compute their union in constant time (line 33).

Clauses are considered differently according to the number of negative literals they contain, called their *length* : $\text{len}(C) = \text{Card}\,(\{x \in \mathcal{V} \,|\, \neg x \in C\})$. When their length is 1, they are *short clauses* else they are *long clauses*. Since the formula can be seen as a bipartite graph, and contains no positive unit clause, clauses are accessed through the *negative* literals they hold. We define $\text{shortCl}(x)$ as the set of short clauses holding $\neg x$ and $\text{longCl}(x)$ as the set of long clauses holding $\neg x$. To each clause C we associate a variable $\text{firstVisit}(C)$ that is the first *visited* variable x such that C holds $\neg x$, and is initialized to $\bot$. Notice both sets $\text{shortCl}(x)$ and $\text{longCl}(x)$ can be implemented as doubly-linked lists of pointers on clauses, this way removing an element from such a list or concatenating two disjoint lists — reflecting the union of the disjoint sets they represent — can be done in *constant* time.

Algorithm 3 describes how to use and maintain this representation on the critical part of the algorithm; it is easy to see how to adapt the rest of the

algorithm with this notation. We assume MakeSet(x) was already called for every $x \in \mathcal{V}$ during the initialisation, i.e. in the function EnumHorn (see algorithm 1).

Now we want to show how we can manage to proceed to union (line 33) of the clauses holding some $\neg x$ and the clauses holding some $\neg y$ in constant time when collapsing a circuit. *Trick3* consists, for every considered variable x, in considering every clause holding $\neg x$, and, if not already visited, in marking it as "first visited by x". If it was already visited, then it was by some y, previously visited, that means with a path from x to y in $G(F)$ and therefore such that the implication $y \to x$ holds by transitivity. The clause C is in the form $\neg x \vee \neg y \vee C'$. We can perform unit resolution of the "clauses" C and $\neg y \vee x$, and deduce $\neg y \vee C'$, which subsumes C. We can therefore replace C by $\neg y \vee C'$ without affecting the set of models of the formula. This last clause does not hold $\neg x$, and has one negative literal less than C. There only remains to manage the case this clause has become a short clause, which is done in lines 22–25.

References

[1] Kenneth A. Berman, John V. Franco, and John S. Schlipf. Unique satisfiability of horn sets can be solved in nearly linear time. *Discrete Applied Mathematics*, 60(1-3):77–91, 1995.

[2] Etienne Grandjean. Sorting, Linear Time and the Satisfiability Problem. *Annals of Mathematics and Artificial Intelligence*, 16:183–236, 1996.

[3] Michel Minoux. The unique horn-satisfiability problem and quadratic boolean equations. *Annals of Mathematics and Artificial Intelligence*, 6:253–266, 1992. 10.1007/BF01531031.

[4] Robert E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975.

[5] Leslie G. Valiant. The complexity of enumeration and reliability problems. *SIAM J. Comput.*, 8(3):410–421, 1979.